ANL-21/55

Argonne NATIONAL LABORATORY

Scalable Interpolation on GPUs for Thermal Fluids Applications

Mathematics and Computer Science

ANL-21/55

Scalable Interpolation on GPUs for Thermal Fluids Applications

prepared by

Neil Lindquist[1,2], Paul Fischer[1,3,4], Misun Min[1]

[1] Mathematics and Computer Science Division, Argonne National Laboratory
[2] Department of Computer Science, University of Tennessee, Knoxville
[3] Department of Computer Science, University of Illinois, Urbana Champaign
[4] Department of Mechanical Science & Engineering, University of Illinois, Urbana Champaign

Contents

Executive Summary

The need for efficient interpolation arises in many use cases for spectral element simulations. Towards this end, we present a fast and robust GPU-enable interpolation utility for NekRS, a spectral-element, Navier-Stokes solver. This utility is based on the `findpts` and `findpts_eval` routines from the *gslib* library. To demonstrate the functionality, we added support for particle tracking. The GPU implementation of the interpolation allows for tracking 100 times more particles for the same computational cost.

1 Introduction

Most computational fluid dynamics (CFD) codes are based on Eulerian representations, where the equations are discretized in a fixed frame of reference through which the fluid flows. A common aspect of this approach is that the solution is typically represented on a fixed grid, or mesh, comprising nodes where solution values are represented. Differential operators are approximated by relationships between neighboring nodal values. These approaches can be quite efficient both for forward operator evaluation (e.g., differentiation) and for system solution (e.g, solving a Poisson problem, which is usually done iteratively for 3D problems). Post-processing quantities, such as drag, heat-transfer coefficients, and turbulence statistics, are also readily accessible with this Eulerian-based representation as one can use the mesh-based representation to evaluate gradients, boundary integrals, and volume integrals.

There are many instances, however, where there is a distinct need to evaluate *off grid* quantities. Applications include overset grids [1, 2, 3], particle tracking (Sec. 5), and numerous solution interrogation techniques such as profile plotting and 2D slices through the domain for visualization or spectral analysis of turbulent flows. Such applications require general, unstructured, interpolation from a predefined set of mesh points, $\mathbf{x}_i$, $i = 1, \ldots, n$ to an arbitrary array of values, $\underline{\tilde{x}}^* = [\mathbf{x}_1^* \ \mathbf{x}_2^* \ \ldots \ \mathbf{x}_m^*]^T$, where $\mathbf{x}_j^* = [x_j^* \, y_j^* \, z_j^*]$. In the case of nodal bases, we have that, for any scalar solution field $u(\mathbf{x})$, $u(\mathbf{x}_i) = u_i$. Evaluation of $u(\mathbf{x}_j^*)$ requires interpolation from "nearby" points, $\mathbf{x}_i$, that are in the neighborhood of $\mathbf{x}_j^*$. The basic challenge for unstructured CFD discretizations is to be able to quickly identify the neighborhood of $\mathbf{x}_j^*$, including the processor to which this neighborhood (in the Eulerian description) is assigned, and to rapidly evaluate the interpolant.

We describe herein a multi-GPU-based interpolation scheme for spectral elements that has been developed in the context of open-source code, NekRS [4]. We address GPU performance issues, parallel scalability, and relative per-point costs of general interpolation versus Navier-Stokes time advancement. We illustrate the utility of this routine for an overset grid application and for Lagrangian particle tracking. A common question in this context is, *If one has an n-point Navier-Stokes solution, how much more expensive is it to also advect n particles?* We address this question in Section 5.

Our focus here is on the general interpolation problem for parallel implementations of the spectral element method (SEM), with a particular emphasis on GPU-based implementations. The SEM [5] is essentially a high-order finite element method restricted to (curvilinear) hexahedral elements within a domain Ω comprising the union of nonoverlapping elements, Ω^e, $e = 1, \ldots, E$. In d space dimensions, each element is the image of the reference element, $\hat{\Omega} := [-1, 1]^d$, given by (in the case $d = 3$),

$$\mathbf{x}^e(\mathbf{r}) \quad = \quad \sum_{k=0}^{N} \sum_{j=0}^{N} \sum_{i=0}^{N} l_i(r) l_j(s) l_k(t) \mathbf{x}_{ijk}^e. \tag{1}$$

Here, the $l_i(r)$ denote Nth-order Lagrange cardinal polynomials satisfying $l_i(\xi_j) = \delta_{ij}$ on the $N + 1$ Gauss-Lobatto-Legendre (GLL) quadrature points, $\xi_j \in [-1, 1]$. The coefficients $\mathbf{x}_{ijk}^e$ denote the mapping of the tensor-product GLL triplets (ξ_i, ξ_j, ξ_k) to Ω^e. Solution and data fields have a similar

description,

$$u(\mathbf{x})|_{\Omega^e} \quad = \quad u^e(\mathbf{r}) = \sum_{k=0}^{N}\sum_{j=0}^{N}\sum_{i=0}^{N} l_i(r)l_j(s)l_k(t)u^e_{ijk}, \tag{2}$$

such that $u(\mathbf{x})$ is defined by the relationships (1)–(2).

In a distributed-memory parallel computing context there is another important index, namely, the MPI rank p to which a given subset of the elements is assigned. Because of the processor-local private-memory model, elements are implicitly indexed by the pair (p, e) for MPI ranks $p = 0, \ldots, P-1$, and local element numbers $e = 1, \ldots, E_p$. In order to evaluate (2) at $\mathbf{x}^*$ one must therefore first identify the full set of coordinates, $(p^*, e^*, \mathbf{r}^*)$, such that *on rank* p,

$$\mathbf{x}^e(\mathbf{r}^*) \quad = \quad \mathbf{x}^*, \tag{3}$$

with $\mathbf{r}^* := [r^*\, s^*\, t^*]$. We label this process, *findpts()*, after the name of the utility that solves (3), as described below. With $(p^*, e^*, \mathbf{r}^*)$ thus determined, one can solve the interpolation problem,

$$u(\mathbf{x}^*) \;=\; u^e(\mathbf{r}^*), \tag{4}$$

for any scalar field $u(\mathbf{x})$. We refer to (4) as *findpts_eval()* and note that it is equally well defined for vector fields, which better amortize the overhead of the initial *findpts()* query.

The remainder of this report is organized as follows. We begin with a contextual overview of the SEM and NekRS in the next section to provide a performance metric for our interpolation routine. Section 3 describes the algorithms and implementation of our interpolation routines. Finally, in Sections 4 and 5, we demonstrate the usage and performance of our routines in two applications, overset grids and particle tracking, respectively.

2 Formulation

Our focus is on the incompressible Navier-Stokes equations (NSE) for velocity ($\mathbf{u}$) and pressure (p),

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u} = \frac{1}{Re}\nabla^2\mathbf{u} - \nabla p,\ \nabla\cdot\mathbf{u} = 0, \tag{5}$$

where $Re = UL/\nu \gg 1$ is the Reynolds number based on flow speed U, length scale L, and viscosity ν. From a computational standpoint, the long-range coupling of the incompressibility constraint, $\nabla\cdot\mathbf{u} = 0$, makes the pressure substep intrinsically communication intensive and a major focus of our effort as it consumes 60-80% of the run time. This system is typically solved with conjugate gradient or GMRES iteration using some type of p-multigrid for preconditioning [6, 7].

The use of iterative solvers obviates the need for matrix construction as one only requires matrix-vector products. The form of the SEM bases (2) allows all operator evaluations to be expressed as *fast tensor contractions*, which can be implemented as BLAS3 operations[1] in only $O(nN)$ work and $O(n)$ memory references, where $n \approx EN^3$ is the number of gridpoints for the

[1]For example, with $\hat{D}_{il} := \frac{dh_l}{dr}|_{\xi_i}$, the first derivative takes the form $u_{r,ijk} = \sum_l \hat{D}_{il}u_{ljk}$, which is readily implemented as an $N_p \times N_p$ matrix times an $N_p \times N_p^2$ matrix, with $N_p = N+1$.

velocity and pressure [8, 9]. This low complexity is in sharp contrast to the $O(nN^3)$ work and storage complexity that would be realized if the system matrices were explicitly formed. Moreover, the C^0 continuity requirements of the nodal bases implies that the SEM is *communication minimal:* data exchanges have unit-depth stencils, independent of N. Finally, local i-j-k indexing avoids much of the indirect addressing associated with fully unstructured approaches, such that high-order SEM implementations can realize significant throughput (e.g., 1.2 FP64 TFLOPS for application of a Poisson operator with $N = 7$ on a single NVIDIA V100 [10]).

NekRS is a relatively new GPU-oriented version of the open source SEM code, Nek5000. It's GPU kernels are written in the open concurrent compute abstraction (OCCA) for cross-platform portability [11]. Many of the kernels originate from the high-performance libParanumal library of [12], which is a fast GPU-oriented library for high-order methods applied to a variety of PDEs. With NekRS it is possible to advance a 51-billion-point NSE solution in < 0.25 seconds per step on 27648 V100s of OLCF's Summit for a complex pebble-bed reactor geometry similar to that shown in Fig. 1, save that it has 352K pebbles.

We contrast the relatively low $O(nN)$ and $O(n)$ complexities of the SEM operator evaluation with that required for general interpolation given by (2). We note that $l_i(r^*)$, $i = 0, \ldots, N$ can be evaluated in $O(N)$ time by exploiting common factors in l_i and l_j for all (i, j) pairs. So, once $\{l_i(r^*)\}$, $\{l_i(s^*)\}$, and $\{l_i(t^*)\}$ are known for $i = 0, \ldots, N$, there are $2(N+1)^3+2(N+1)^2+2(N+1) = O(N^3)$ operations required to successively condense out the i, then j, then k index in u^e_{ijk}. A critical observation is that, if one has n points, $\mathbf{x}^*_j$, then the total cost for *findpts_eval()* is $O(nN^3) = O(EN^6)$, which is significantly larger (modulo constants) than the SEM time-advancement of the NSE. Moreover, *findpts()* is generally much more expensive than *findpts_eval()*, so the constants for interpolation are not small if one has moving interrogation points, as is the case for Lagrangian particle tracking. It is therefore imperative to have a fast interpolation routine if one wants to evaluate the solution at a relatively large number of interpolation points.

We point out that our new GPU-based *findpts()* utilities derive from CPU-oriented C-code in the *gslib* library written originally by James Lottes [13]. *gslib* is a lightweight C communication package that readily links with any Fortran, C, or C++ code. Its interpolation routines grew out of a need to support data interrogation and Lagrangian particle tracking on $P = 10^4$–10^6 processors. High-fidelity interpolation for highly curved elements, like the ones supported by the SEM, is quite challenging. Thus, *findpts* was designed with the principles of robustness (i.e., it should never fail) and speed (i.e., it should be fast). However, *gslib* was developed before the rise of GPU-accelerated supercomputers. So, it's performance is significantly limited by its inability to access most of the computational power of the many current and upcoming heterogeneous supercomputers [14]. Furthermore, using the original C-code would require copying the field data ($\{u^e_{ijk}\}$) to be copied to the host, further reducing performance.

3 Interpolation Algorithm

The interpolation is composed of two stages. First, the *findpts* routine takes a point in physical space and computes the corresponding element and the coordinates in the reference element. Second, the *findpts_eval* routine takes this position information and interpolations a field at each of those points using (2).

Algorithm 1 `findpts_local`

```
 1: procedure OGS_FINDPTS_LOCAL(code, el, r, dist2, x, npt)
 2:     for i from 0 to npt do
 3:         elements ← hashtable_lookup(i)
 4:         for element ∈ elements do
 5:             if x ∈ bounding_box(element) then
 6:                 for j = 1, 2, ..., 50 do
 7:                     Compute residual ρ_j, Jacobian J, Hessian H
 8:                     if ‖ρ_j‖ < tol then
 9:                         exit loop
10:                     else if ρ_j > ρ_{j−1} then
11:                         Reduce trust region t
12:                         r ← r − u
13:                     else
14:                         Compute u to minimize ‖ρ_j − Ju‖_2
                                  st |u| ≤ t and |r + u| ≤ 1
15:                         r ← r + u
16:                     end if
17:                 end for
18:             end if
19:         end for
20:     end for
21: end procedure
```

3.1 findpts

The *findpts* routine first attempts to find the points among its local elements through the internal routine *findpts_local*. Then, candidate processes are computed for all unfound points. Those processes each attempt to find the point among its local element, again using *findpts_local*. The results are returned to each originating process where the best result is kept.

The *findpts_local* routine works in a similar manner. For each point, a set of candidate elements are computed. A reference coordinate is found for each candidate element using the *findpts_el* routine. The most accurate coordinate is kept for each point.

Recall that the mapping from the reference element, $\hat{\Omega} = [-1, 1]^3$, to each element, Ω^e, is defined by (2). So, the *findpts_el* routine uses Newton iterations to find the coordinates in $\hat{\Omega}$ that map to the given point. The distance to each GLL point is computed, with the closest being used as the initial guess. During the Newton solve, each step is restricted to a trust region and the new solution is restricted to $\hat{\Omega}$. The trust region starts as a cube of length 2. If any step increases the residual, the trust region is halved in each dimension and the previous solution is restored.

Hash tables are used by *findpts* and *findpts_local* to compute candidate processes, and elements, respectively. For each table, a uniform, axis-aligned grid is placed over the mesh (either global or process-local). The global hash table records for each grid cell the processes which contain elements overlapping that cell, whereas the local hash tables record for each grid cell the elements overlapping that cell. To prevent points from falling between elements, due to rouding errors, each element is slightly expanded when testing overlap. The uniformity of the grid simplifies the computation of candidate processes or elements. During lookup, the local hash table further filters elements by comparing the point to the expanded element's axis-aligned bounding box.

Algorithm 2 `findpts_local_eval`

procedure OGS_FINDPTS_LOCAL_EVAL(out, el, r, npt, in)
  for i from 0 to npt **do**
    for $j = 0, \ldots, N$ **do**
      $\text{wtr}[j] = \text{CARDINAL\_EVAL}(0, r[i][0], j)$
      $\text{wts}[j] = \text{CARDINAL\_EVAL}(0, r[i][1], j)$
      $\text{wtt}[j] = \text{CARDINAL\_EVAL}(0, r[i][2], j)$
    end for
    for $j = 0, \ldots, N$ **do**
      $s[j] \leftarrow \text{wtr}[j] \sum_{k=0}^{N} \text{wts}[k] \sum_{\ell=0}^{N} \text{wtt}[\ell] \cdot \text{in}[el, j, k, \ell]$
    end for
    $\text{out}[i] \leftarrow \sum_{j=0}^{N} s[j]$
  end for
end procedure

We implemented *findpts_local* and *findpts_el* in OCCA as a single kernel. Each point is given a seperate outer iteration (e.g. one thread-block in CUDA) in order to reduce branch divergence between the inner iterations, important for GPU backends. This differs from *gslib* which processes all points for a candidate element together. Our implementation is otherwise similar to the implementations in *gslib*. The inner iterations (e.g. threads in CUDA) are used to take advantage of parallelism such as computation of cardinal polynomials and tensor products. Algorithm 1 shows an outline of our formulation.

3.2 findpts_eval

The *findpts_eval* routine consists of an all-to-all exchange based on the processes of the elements found by *findpts*, a tensor product for each point using *findpts_local_eval*, then returning the results with another all-to-all exchange. Similar to *findpts*, we implemented this in OCCA using a single kernel to handle the entire local evaluation. Furthermore, we assign one outer iteration (e.g. one thread-block in CUDA) per interpolation point. The inner iterations (e.g. threads in CUDA) parallelized the computation of the cardinal polynomials and the tensor product. Algorithm 2 shows our formulation.

3.3 Lagrange Cardinal Polynomials

Both of these routines must evaluate the Lagrange cardinal polynomials, and possibly it's derivatives, at given points. This behavior is encapsulated in CARDINAL_EVAL. Our implementation is inspired by the $\mathcal{O}(n)$ serial algorithms, modified to use multiple GPU threads. Is is written in a SIMD manner where each inner iteration computes the value of a single cardinal polynomial. Furthermore,

Algorithm 3 `cardinal_eval`

```
1: procedure CARDINAL_EVAL(nderiv, x, j)
2:     if i < n then
3:         z ← lookup quadrature points
4:         w ← lookup weights
5:         if nderiv = 2 then
6:             u_0 = 1, u_1 = 0, u_2 = 0
7:             for k = 0, ..., N do
8:                 if i ≠ j then
9:                     d_j = 2(x − z_k)
10:                    u_2 ← u_2 · d_k + u_1
11:                    u_1 ← u_1 · d_k + u_0
12:                    u_0 ← u_0 · d_k
13:                end if
14:            end for
15:            p_0[j] =   w[j] · u_0
16:            p_1[j] = 2 · w[j] · u_1
17:            p_2[j] = 8 · w[j] · u_2
18:        else if nderiv = 1 then
19:            ...
20:        else if nderiv = 0 then
21:            ...
22:        end if
23:    end if
24: end procedure
```

we were able to significantly simplify the evaluation by noting that for the ith cardinal polynomial,

$$
\begin{aligned}
d_j &= x - z_j \\
u_0^{(j)} &= \prod_{\substack{q=0 \\ q\neq i}}^{j} d_q && = d_j u_0^{(j-1)} \\
u_1^{(j)} &= \sum_{\substack{p=0 \\ p\neq i}}^{j} \prod_{\substack{q=0 \\ q\neq i,p}}^{j} d_q && = d_j u_1^{(j-1)} + u_0^{(j-1)} \\
u_2^{(j)} &= \sum_{\substack{o=0 \\ o\neq i}}^{j} \sum_{\substack{p=0 \\ p\neq i,o}}^{j} \prod_{\substack{q=0 \\ q\neq i,o,p}}^{j} d_q = d_j u_2^{(j-1)} + 2u_1^{(j-1)}.
\end{aligned}
$$

Then, the cardinal polynomial and its first two derivatives are equal to $w_0 u_0^{(N)}$, $w_1 u_1^{(N)}$, and $w_2 u_2^{(N)}$, respectively, where w_0, w_1, w_2 are normalization constants for that polynomial. This allows the polynomial and its derivatives to be computed in $\mathcal{O}(N)$ time and constant memory. Furthermore, it helps the performance of GPU backends by ensuring each thread accesses the same values from memory and limiting thread divergence. Algorithm 3 shows our formulation for second derivative evaluation; lower derivative evaluations just remove the unneeded terms.

Algorithm 4 Overlapping Meshes Outline

```
for timestep do
    if mesh is moving then
        Find new boundary interpolation points
    end if
    Interpolate overlapped boundary values
    Solve fluid equations
end for
```

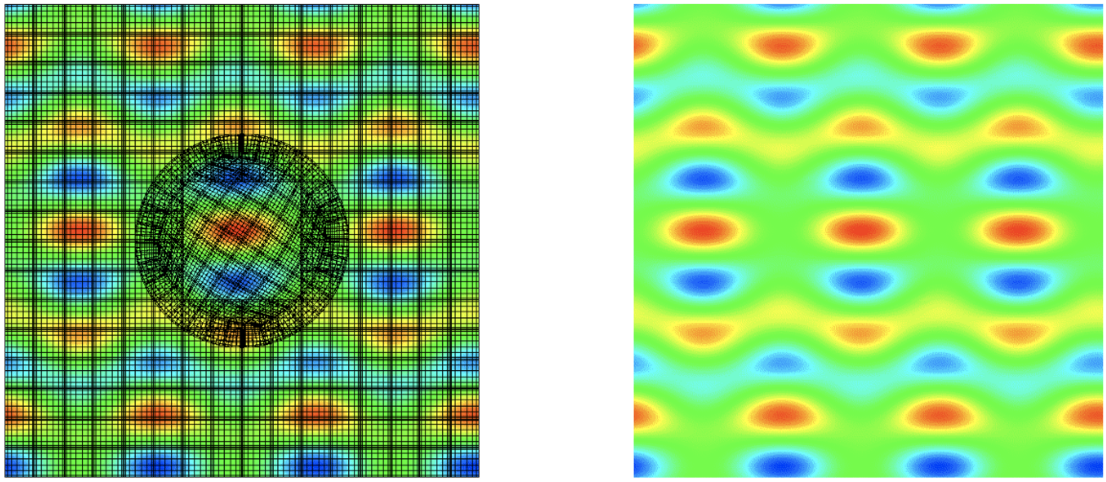

Figure 1: 3D NekRS neknek simulation with two meshes, a rotating cylinder and square annulus shown with (left) and without (right) the spectral element grids. No sign of mesh imprinting is visible in the figure on the right.

4 Applications: Overlapping meshes

Automatic generation of meshes with hex-elements can be difficult for certain complex geometries. One approach to simplifying the meshing process is to use a Schwarz overlapping method to divide the domain into multiple, easier to mesh regions [3]. Furthermore, dividing the domain into multiple meshes can simplify simulations that have moving components by meshing individual parts that only move relative to the other subdomains. Finally, meshes with varying levels of accuracy can be simplified by using overlapping meshes rather than a single mesh.

In the Schwarz overlapping method, the solution in each subdomain can be advanced independently, with boundary conditions in the overlapping regions interpolated from the solution in the overlapping neighbors at the previous time step. This approach provides $O(\Delta t)$ accuracy that can be improved through Picard or predictor-corrector iterations.

Implementation of an overlapped mesh scheme requires interpolating the fluid velocity and any scalars, such as temperature, at each timestep, and multiple times per step if correction iterations are added. Additionally, changes to the mesh during the simulation, such as for moving machinery, requires recomputing the interpolation points at every time step. Thus, using overlapped meshes requires scalable interpolation routines.

Algorithm 5 Particle Update with 3rd order time-stepping

1: FINDPTS($\vec{x}, procs, el, r$)
2: MIGRATE($\vec{x}, procs, el, r, \vec{u}$)
3: FINDPTS_EVAL($\vec{u}_{:,0}, el, r,$ fluid_velocity)
4: **for** $p \in$ particles **do**
5: $\quad \vec{x}_p \leftarrow \vec{x}_p + \Delta t(c_1\vec{u}_{p,0} + c_2\vec{u}_{p,1} + c_3\vec{u}_{p,2})$
6: $\quad \vec{u}_{p,2} \leftarrow \vec{u}_{p,1}; \vec{u}_{p,1} \leftarrow \vec{u}_{p,0}$
7: $\quad$ Check boundary conditions
8: **end for**

The overset grid support requires resolution of several other technical issues relating to mass conservation, timestepping, and parallelism which have been addressed for the SEM in [15, 16] and in earlier efforts by [2, 17]. For a GPU-based NSE solver, however, it is also critical to have fast interpolation on the GPU, such as provided by the routines described in the preceeding section. Figure 1 illustrates results for an initial 3D test case running on a single node of Summit with a rotating cylindrical mesh embedded in a fixed outer mesh. This case is an exact Navier-Stokes solution due to [18]. The two subdomains run as separate MPI processes.

5 Particle Tracking

Tracking particles within a fluid flow occurs in a number of use cases. Of recent importance is the simulation of aerosolized COVID-19 particles [19, 20]. Other applications include river sediment [21] and industrial processes [22]. Furthermore, tracer particles can be used to visualize the fluid's movement [23]. Our tests focused on Lagrange tracer particles where the particle velocity is identical to the surrounding fluid velocity, but this work can be easily extended to inertial particles. Figures 2–3 shows examples of visualizing the fluid with tracer particles.

5.1 Particle Migration

Particles can be easily simulated with an explicit time-stepping scheme, such as Adam-Bashford, with the velocity computed at the particle's old position. We summarize the particle update steps in Algorithm 5. Our particle boundary conditions were implemented on a case-by-case basis for simplicity's sake.

One notable step is that of particle migration which exchanges particle ownership so that each process owns the particles that are present in its elements. This alleviates all communication in the following *findpts_eval* calls. Furthermore, particle will usually spent multiple timesteps within a single element, reducing the communication needed for subsequent *findpts* calls and migrations. The desired ownership of a particle is already computed during *findpts*, so implementing the migration is just a matter of doing an all-to-all exchange on the particles needing migration.

5.2 Performance

Our performance tests focus on the gabls-part example, which is based on the first case of the GEWEX atmospheric boundary layer study (GABLS) as described by [24]. This problem simulates

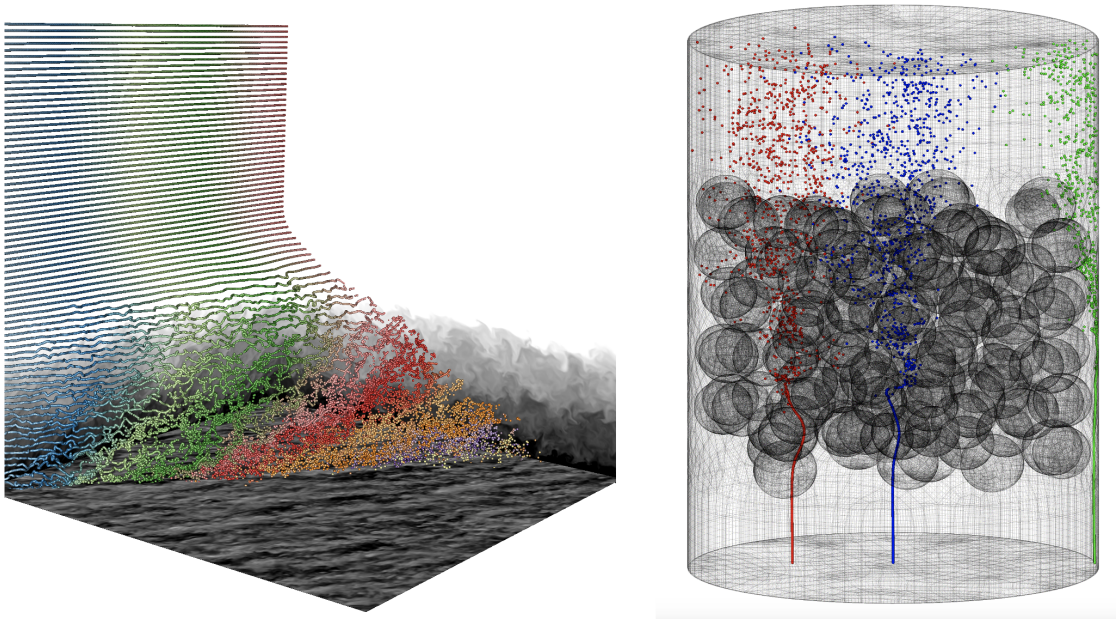

Figure 2: Examples of tracer particles for an atmospheric boundary layer simulation and a pebble bed simulation with 146 pebbles.

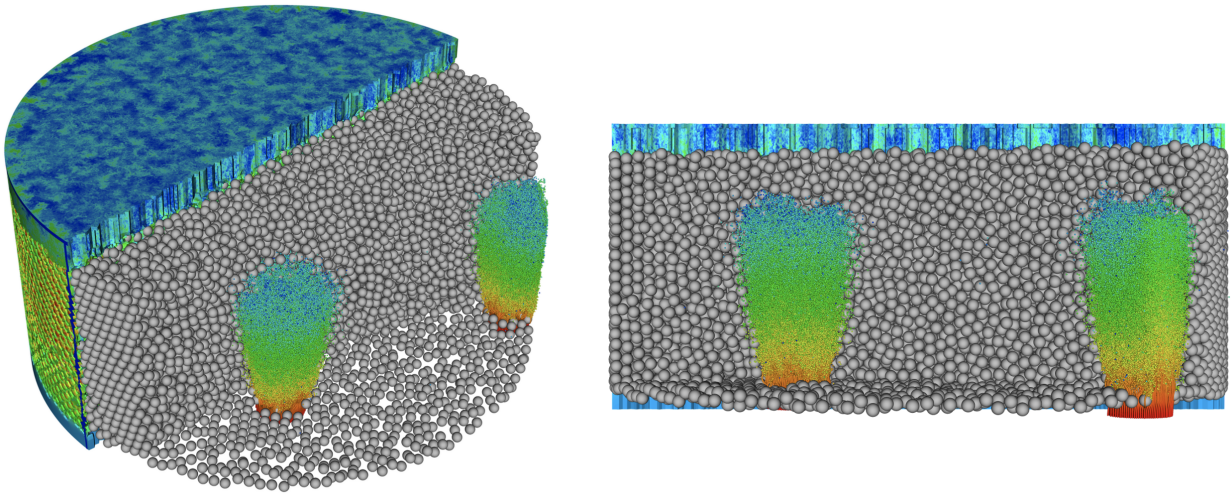

Figure 3: Examples of tracer particles for a pebble bed simulation with 44 257 pebbles.

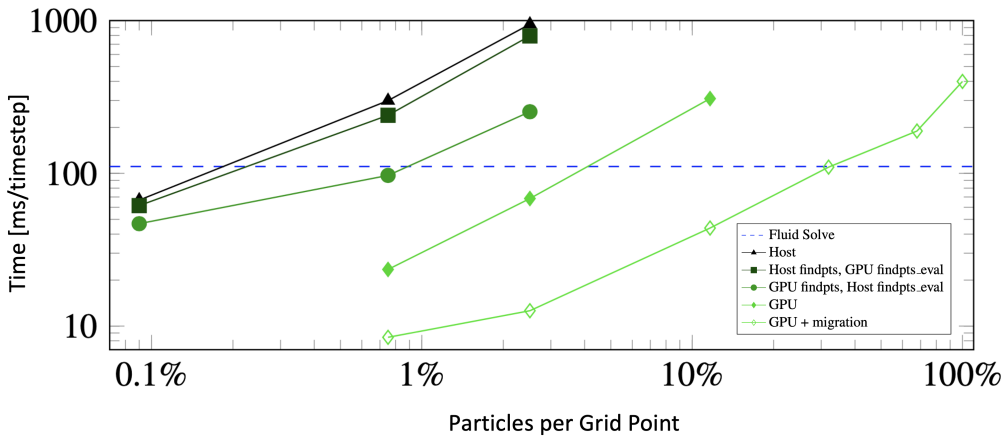

Figure 4: Comparison of the implemented optimizations with $64^3 \times 8^3 \approx 134 \times 10^6$ grid points distributed over 60 GPUs. Solid lines show time to simulate particles in addition to the cost to simulate the fluid.

a region of $400\,\mathrm{m} \times 400\,\mathrm{m} \times 400\,\mathrm{m}$ with periodic boundary conditions on the horizontal axes, vertical velocity of zero lower boundaries, and a free-slip boundary condition on the upper boundary with a vertical velocity of zero and a horizontal velocity of $8\,\mathrm{m/s}$ in the east-west direction. A Coriolis effect corresponding to the latitude 73°N was applied.

Both of the following tests use a mesh of 64^3 cubic elements in each direction with degree 8 polynomials within each element. The problem was then run with 10 nodes of Summit (60 GPUs). The points were evenly distributed in each dimension and assigned to processes in a round robin fashion. Periodic boundary conditions were applied to the particles in the x and z directions (following the velocity BCs). The tests were run for 2000 steps with all particles being created on the first step.

We first tested the performance of various combinations of optimizations, including

- moving *findpts_local* to OCCA,
- moving *findpts_eval_local* to OCCA, and
- enabling particle migration.

Figure 4 shows the wall clock time per timestep in excess of the time to simulate the fluid without any particles. As can be seen in the figure, porting both routines provide significantly more benefit that either routine on its own. The combination of all three optimizations provides a 35× speedup in the particle overhead for 10^3 particles, and a 100× increase in the number of particles with the particle overhead is limited to the cost of the fluid solve. These optimizations put the cost of simulating a particle on par with the cost to simulating just a few fluid grid points; furthermore, it allows for the simulation of over a hundred million of particles across 60 processes.

Next we tested the performance of various components with results shown in Table 1. The three components that stand out are the host *findpts*, the host *findpts_eval*, and copying to velocity to the host; switching to device provides dramatic savings in the first two, and completely removes

Table 1: Cost in ms per timestep of various components in particle tracking. Nested timings are included in their parent times.

Findpts implementation	GPU						CPU			
Migration	Yes			No			Yes		No	
particle count	100^3	150^3	200^3	100^3	150^3	200^3	100^3	150^3	100^3	150^3
Fluid Solve	98.2	98.1	98.5	99.2	98.2	107.5	97.9	98.7	101.2	100.1
Particle Creation	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Particle Update	3.6	8.5	19.1	18.6	60.8	146.1	277.0	910.2	288.3	953.9
- Copy fluid vel. to host	-	-	-	-	-	-	15.7	13.9	16.4	15.5
- findpts	2.7	6.4	14.3	8.9	29.0	69.1	225.3	746.5	234.1	775.9
- - Memcpy	0.4	0.7	1.2	0.8	1.8	4.2	-	-	-	-
- - Kernel	1.8	4.9	11.3	1.9	6.4	14.6	219.7	735.7	220.4	734.7
- migration	0.1	0.1	0.3	-	-	-	0.1	0.2	-	-
- findpts_eval	0.6	1.3	3.1	9.4	31.0	74.9	49.7	158.5	54.0	177.1
- - Memcpy	0.2	0.4	0.9	0.3	1.0	2.0	-	-	-	-
- - Kernel	0.5	0.9	2.2	0.3	0.8	1.4	49.6	158.5	43.2	141.9
- Advance position	0.2	0.5	1.2	0.2	0.5	1.3	0.2	0.5	0.1	0.5
- Barrier	0.0	0.1	0.2	0.1	0.3	0.8	1.8	4.4	0.1	0.3

the last. With the device implementation, particle migration is able to provide significant additional improvements in the communication, resulting in, e.g., 70 % of the total update time being spent in the computational kernel for 200^3 particles instead of merely 11 %.

There are a few points to note when extrapolating these results to other cases. First, the mesh is linear and uniformly sized, meaning that *findpts* will require only one Newton interation to converge. Second, the particles are distributed uniformly; in many real applications the particle distribution will induce load imbalance, which reduces performance. Third, particles were created once at the beginning of the simulation. High particle turnover will produce extra communication in the *findpts* and migration steps, as most particles will require migration on their first timestep. Finally, in the fluid simulation the velocity, pressure, and temperature solves converged within one iteration on almost every timestep. So, these case is an easy problem for both the particle tracking and the fluid solve.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357 and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

The research used resources at the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract DE-AC05-00OR22725 and at the Argonne Leadership Computing Facility, under

Contract DE-AC02-06CH11357.

References

[1] G. Chesshire and W. Henshaw. Composite overlapping meshes for the solution of partial differential equations. *J. Comput. Phys.*, 90:1–64, 1990.

[2] B.E. Merrill, Y.T. Peet, P.F. Fischer, and J.W. Lottes. A spectrally accurate method for overlapping grid solution of incompressible Navier-Stokes equations. *J. Comput. Phys.*, 307:60–93, 2016.

[3] Ketan Mittal, Som Dutta, and Paul Fischer. Nonconforming schwarz-spectral element methods for incompressible flow. *Computers & Fluids*, 191:104237, 2019.

[4] Paul Fischer, Stefan Kerkemeier, Misun Min, Yu-Hsiang Lan, Malachi Phillips, Thilina Rathnayake, Elia Merzari, Ananias Tomboulides, Ali Karakus, Noel Chalmers, and Tim Warburton. Nekrs, a gpu-accelerated spectral element navier-stokes solver. *CoRR*, abs/2104.05829, 2021.

[5] A.T. Patera. A spectral element method for fluid dynamics : laminar flow in a channel expansion. *J. Comput. Phys.*, 54:468–488, 1984.

[6] J. W. Lottes and P. F. Fischer. Hybrid multigrid/Schwarz algorithms for the spectral element method. *J. Sci. Comput.*, 24:45–78, 2005.

[7] Martin Kronbichler and Karl Ljungkvist. Multigrid for matrix-free high-order finite element computations on graphics processors. *ACM Transactions on Parallel Computing*, 6(1):1–32, 05 2019.

[8] M.O. Deville, P.F. Fischer, and E.H. Mund. *High-order methods for incompressible fluid flow.* Cambridge University Press, Cambridge, 2002.

[9] S.A. Orszag. Spectral methods for problems in complex geometry. *J. Comput. Phys.*, 37:70–92, 1980.

[10] Paul Fischer, Misun Min, Thilina Rathnayake, Som Dutta, Tzanio Kolev, Veselin Dobrev, Jean-Sylvain Camier, Martin Kronbichler, Tim Warburton, Kasia Swirydowicz, and Jed Brown. Scalability of high-performance PDE solvers. *The International Journal of High Performance Computing Applications*, 34(5):562–586, 2020.

[11] David S Medina, Amik St-Cyr, and Tim Warburton. OCCA: A unified approach to multi-threading languages. *preprint arXiv:1403.0968*, 2014.

[12] N. Chalmers, A. Karakus, A. P. Austin, K. Swirydowicz, and T. Warburton. libParanumal, 2020.

[13] gslib: Gather-scatter library, 2020.

[14] Awais Khan, Hyogi Sim, Sudharshan S. Vazhkudai, Ali R. Butt, and Youngjae Kim. An analysis of system balance and architectural trends based on Top500 supercomputers. In *The International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2021, pages 11–22, New York, NY, USA, January 2021. Association for Computing Machinery.

[15] Ketan Mittal, Som Dutta, and Paul Fischer. Nonconforming Schwarz-spectral element methods for incompressible flow. *Computers and Fluids*, 191, 2019.

[16] Ketan Mittal, Som Dutta, and Paul Fischer. Multirate time-stepping for the incompressible Navier-Stokes equations in overlapping grids. *jcp*, 437:110335, 2020.

[17] B.E. Merrill and Y.T. Peet. Moving overlapping grid methodology of spectral accuracy for incompressible flow solutions around rigid bodies in motion. *J. Comput. Phys.*, 390:121–151, 2019.

[18] O. Walsh. Eddy solutions of the navier-stokes equations. In J.G. Heywood, K. Masuda, R. Rautmann, and V.A. Solonikkov, editors, *The NSE II-Theory and Numerical Methods*, pages 306–309. Springer, 1992.

[19] Mahshid Mirzaie, Esmail Lakzian, Afrasyab Khan, Majid Ebrahimi Warkiani, Omid Mahian, and Goodarz Ahmadi. COVID-19 spread in a classroom equipped with partition – A CFD approach. *Journal of Hazardous Materials*, 420:126587, October 2021.

[20] Jana Wedel, Paul Steinmann, Mitja Štrakl, Matjaž Hriberšek, and Jure Ravnik. Can CFD establish a connection to a milder COVID-19 disease in younger people? Aerosol deposition in lungs of different age groups based on Lagrangian particle tracking in turbulent flow. *Computational Mechanics*, 67(5):1497–1513, May 2021.

[21] Som Dutta, Mark W. Van Moer, Paul Fischer, and Marcelo H. Garcia. Visualization of the Bulle-Effect at River Bifurcations. In *Proceedings of the Practice and Experience on Advanced Research Computing*, PEARC '18, pages 1–4, New York, NY, USA, July 2018. Association for Computing Machinery.

[22] Susana Torno, Javier Toraño, and Inmaculada Álvarez-Fernández. Simultaneous evaluation of wind flow and dust emissions from conveyor belts using computational fluid dynamics (CFD) modelling and experimental measurements. *Powder Technology*, 373:310–322, August 2020.

[23] Tamay M. Özgökmen and Paul F. Fischer. CFD application to oceanic mixed layer sampling with Lagrangian platforms. *International Journal of Computational Fluid Dynamics*, 26(6-8):337–348, July 2012.

[24] Robert J Beare, Malcolm K Macvean, Albert AM Holtslag, Joan Cuxart, Igor Esau, Jean-Christophe Golaz, Maria A Jimenez, Marat Khairoutdinov, Branko Kosovic, David Lewellen, et al. An intercomparison of large-eddy simulations of the stable boundary layer. *Boundary-Layer Meteorology*, 118(2):247–272, 2006.

Mathematics and Computer Science
Argonne National Laboratory
9700 South Cass Avenue, Bldg. 240
Lemont, IL 60439

www.anl.gov

Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC